POWER CHALLENGE 10000

Leading the HPC Revolution

SiliconGraphics
Computer Systems

MIPS R10000 Leadership

High Performance Integer & Floating Point

Applicable to the spectrum of computing problems

High Frequency Single Chip Implementation

Low cost
High volume

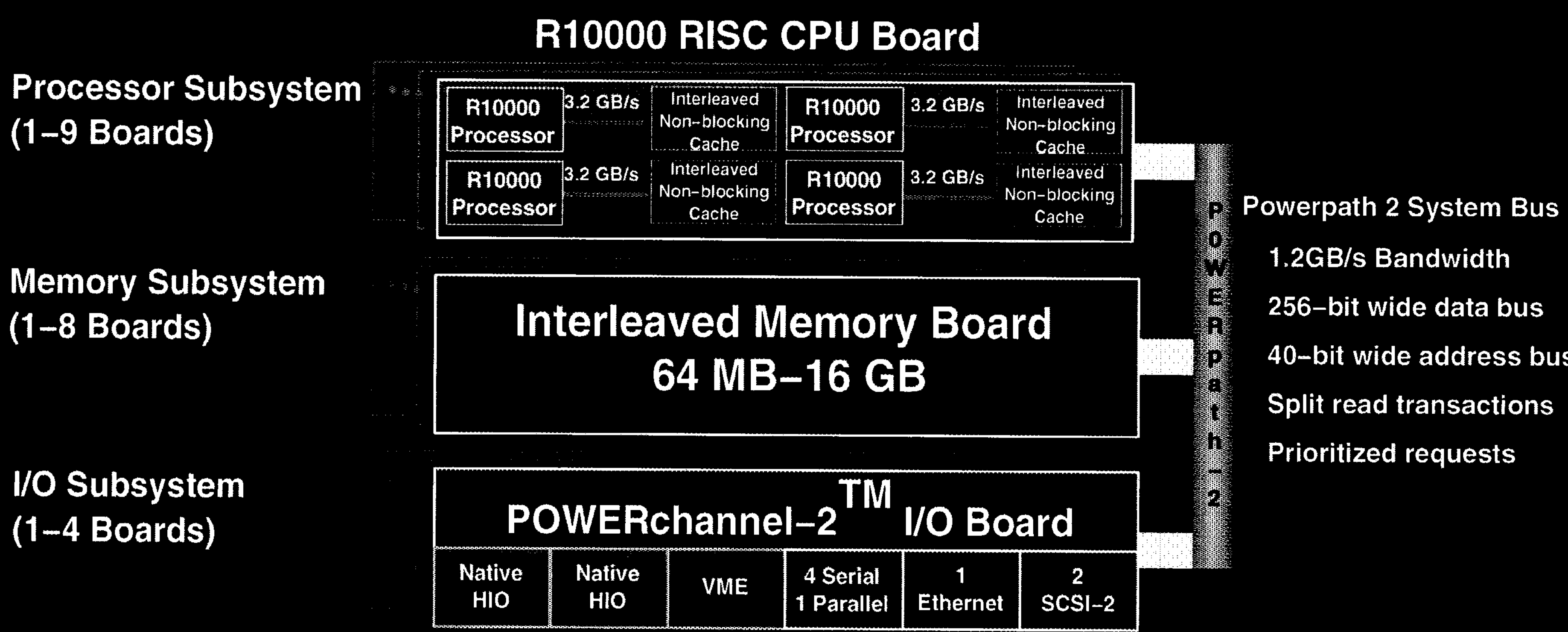
Latency–Tolerant Architecture

Performance can be realized on non–optimized codes

High Bandwidth Memory Interface

Performance can be realized on large, real–world problems

POWER CHALLENGE 10000 Architecture



SiliconGraphics
Computer Systems

MIPS R10000 Performance Technologies

Superscalar Architecture

- Four instruction/cycle
- 2 integer + 2 floating pt. + 1 load/store unit

→ **More processing in less time**

High Performance Cache

- 1MB L2 cache
- Dedicated 3.2GB cache bus
- Interleaved cache access
- Non–blocking cache

→ **Less time waiting for scattered data**

Out–of–Order Execution

- 3 instruction queues
- Up to 32 instructions in progress simultaneously
- 64 physical 64–bit registers with renaming

→ **Existing binaries run faster**

Branch Prediction

- Speculative execution
- Up to 4 outstanding branch predictions

→ **Dusty deck codes run faster**

POWER CHALLENGE 10000 XL

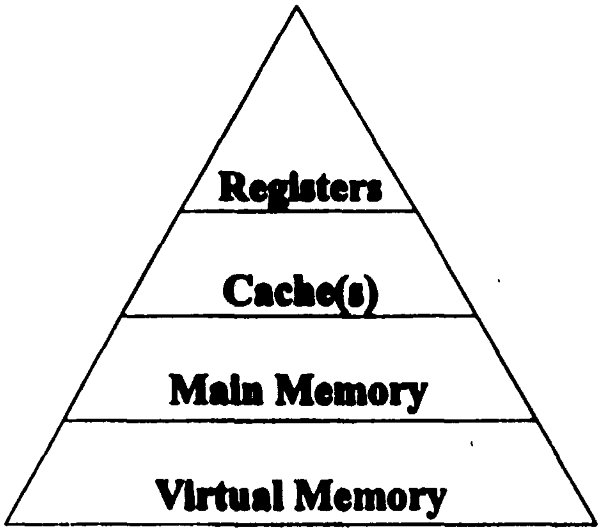
Highly Scalable Interactive Supercomputer

- 2–36 R10000 CPUs
- 1 MB Secondary Cache Per CPU
- Over 14 Peak GFLOPS
- 1.2 GB/sec System Bus
- 64 MB – 16 GB RAM, 1, 2, 4, or 8–way interleaved
- 2 GB – 68.8 GB Disk (8.2 TB External RAID)*
- Optional Viz Console

*assumes 4.3 GB disk drives

SiliconGraphics
Computer Systems

Background Information



- Modern RISC systems use a hierarchy of memory systems, which tradeoff cost vs. speed vs. size.
- In order to achieve the best possible level of performance, one must maximize the level of data reuse.
- Most RISC systems can perform at most one memory operation (load/ store) per floating point instruction, without a loss of performance.
- The Cray C90 can perform three memory operations (two loads and one store) per chained pair of multiply and add instructions.

Why do Machines have Memory Hierarchies?

To optimize price−performance given the widening gap between CPU and memory performance.

To exploit increased density of microprocessor technology by integrating memory onto the chip.

The Cache Design Approach

Use fast/expensive SRAM or on−chip real−estate to implement small caches with high bandwidth and low latency access.

Use slower/inexpensive DRAM to implement large main memory with lower bandwidth and higher latencies for access.

Transfer data or instructions on demand into cache. Retain in cache until the space is needed for newly−demanded data/instructions.

Analogous to virtual memory and demand paging between RAM and disk, but implemented in hardware rather than the operating system.

When are Caches most Effective?

When programs exhibit data/instruction locality.

- *Temporal* locality: If data/instructions are referenced, they will be referenced again soon.

- *Spatial* locality: If data/instructions are referenced, nearby data/instructions will be referenced soon.

Many programs contain extensive locality, and automatic compiler optimizations or manual algorithmic improvements can increase locality and cache effectiveness.

Power Challenge Tuning

Step 1: Get the right answers

Step 2: Use existing tuned code

libfastm
libcomplib.sgimath

Step 3: Get the loops to Software Pipeline

Use prof to identify important loops
Compile –O3
Read the compiler <swp> messages
Register blocking/outer loop unrolling
IU–FPU latency
Inlining
Loop splitting
Compiler options
C loops

Step 4: Live dangerously

–OPT:roundoff=3
–OPT:IEEE_arithmetic=3
–TENV:X=4
–GCM:...speculation
Arithmetic reassociation

Step 5: Modify code for better cache utilization

Use pixie to identify problem areas
Exploit locality
Cache thrashing and array padding
Loop fusion
Blocking

Get the Right Answers

Many codes will port with a simple recompilation

Try porting to –O2 –mips4

Sometimes they don't

64–bit processor & OS
longs & pointers are 64 bits
ints are still 32 bits

Another vendor's libraries

Standards violations
–static –O0 may forgive some in FORTRAN

Mistakes

Use prof to Know Where to Tune

PC–sampling profiling:

Program counter location recorded every 10ms

Provides sorted list of time spent in each subroutine,
line level profiling options

Works on MP programs, too

Times reported reflect true runtime of program
- Cache misses
- Bank conflicts
- Load imbalance

No need to recompile, just re–link

```
% ld  -p -o program ...
% cc  -p -o program ...
% f77 -p -o program ...
```

`% program` (creates mon.out)

```
% prof [-heavy -lines] program
```

prof Output

```
--------------------------------------------------------------------------------
Profile listing generated Thu Dec  1 11:13:23 1994
    with:       prof adi2.p
--------------------------------------------------------------------------------

samples   time    CPU    FPU   Clock    N-cpu  S-interval Countsize
   1196    12s  R8000  R8010  75.0MHz    0     10.0ms     0(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.08% of 11.9600sec)


--------------------------------------------------------------------------------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
--------------------------------------------------------------------------------

samples   time(%)       cum time(%)      procedure (file)

    833   8.3s( 69.6)  8.3s( 69.6)          ZSWEEP (adi2.p:.../adi2.f)
    108   1.1s(  9.0)  9.4s( 78.7)          YSWEEP (adi2.p:.../adi2.f)
    101     1s(  8.4)   10s( 87.1)          XSWEEP (adi2.p:.../adi2.f)
     49  0.49s(  4.1)   11s( 91.2)          irand_ (/usr/lib64/libftn.so:.../rand_.c)
     46  0.46s(  3.8)   11s( 95.1)             ADI (adi2.p:.../adi2.f)
     40   0.4s(  3.3)   12s( 98.4)           rand_ (/usr/lib64/libftn.so:.../rand_.c)
     14  0.14s(  1.2)   12s( 99.6)    ADI.PREGION1 (adi2.p:.../adi2.f)
      2  0.02s(  0.2)   12s( 99.7)    ADI.PREGION0 (adi2.p:.../adi2.f)
      1  0.01s(  0.1)   12s( 99.8)         _syssgi (/usr/lib64/libc.so.1:.../syssgi.s)
      1  0.01s(  0.1)   12s( 99.9)        t_delete (/usr/lib64/libc.so.1:.../malloc.c)
      1  0.01s(  0.1)   12s(100.0)    _sigprocmask (/usr/lib64/libc.so.1:.../possig.s)

   1196    12s(100.0)   12s(100.0)           TOTAL
```

Use Existing Tuned Code

libfastm

sin, cos, tan, pow, exp, log, cis

Big performance gain traded for slightly less accuracy

```
f77 -o prog prog.o -lfastm [-lm]
```

libcomplib.sgimath

Versions for –mips1, –mips2, –mips3, –mips4

BLAS Levels 1, 2 and 3
EISPACK (Not tuned)
LINPACK (Not tuned)
LAPACK
FFTs & Convolutions
SLATEC (Not tuned)

```
f77 -o prog prog.o -lcomplib.sgimath

f77 -mp -o prog prog.o -lcomplib_mp.sgimath
```

Register Blocking

Outer Loop Unrolling: reduces loads of a by nb

```
      subroutine mm1(a,lda,b,ldb,c,ldc,m,l,n)
      integer lda, ldb, ldc, m, n
      real*8 a(lda,l), b(ldb,n), c(ldc,n)
c
      do j = 1, n, nb
         do k = 1, l
            do i = 1, m
               c(i,j+0)    = c(i,j+0)    - a(i,k)*b(k,j+0)
               c(i,j+1)    = c(i,j+1)    - a(i,k)*b(k,j+1)
                                    .
                                    .
                                    .
               c(i,j+nb-1) = c(i,j+nb-1) - a(i,k)*b(k,j+nb-1)
            enddo
         enddo
      enddo
c
      return
      end
```

Middle Loop Unrolling: reduces ld/st of c by lb

```
      subroutine mm2(a,lda,b,ldb,c,ldc,m,l,n)
      integer lda, ldb, ldc, m, n
      real*8 a(lda,l), b(ldb,n), c(ldc,n)
c
      do j = 1, n
         do k = 1, l, lb
            do i = 1, m
               c(i,j) = c(i,j) - a(i,k+0)*b(k+0,j)
               c(i,j) = c(i,j) - a(i,k+1)*b(k+1,j)
                                    .
                                    .
                                    .
               c(i,j) = c(i,j) - a(i,k+lb-1)*b(k+lb-1,j)
```

Play into Known Optimizations

Use reciprocal–square–root (with –OPT:IEEE_arithmetic=3)

```
p2 = x*x / y
p = sqrt(p2)
```

should instead be written as:

```
p = abs(x) * ( 1.0 / sqrt(y) )
p2 = p*p
```

Split transcendental functions into vector–style loops

```
do i=1,n
   compute x(i)
enddo

do i=1,n
  y(i) = exp(x(i))
enddo

do i=1,n
   use y(i)
enddo
```

because
(1) non–transcendental loops will SWP, and
(2) with upcoming compiler, vector intrinsics will be used.

Loop Splitting

```
do i=lft,llt
  x17(i)=x7(i)-x1(i)
  x28(i)=x8(i)-x2(i)
  x35(i)=x5(i)-x3(i)
  x46(i)=x6(i)-x4(i)
  y17(i)=y7(i)-y1(i)
  y28(i)=y8(i)-y2(i)
  y35(i)=y5(i)-y3(i)
  y46(i)=y6(i)-y4(i)
  z17(i)=z7(i)-z1(i)
  z28(i)=z8(i)-z2(i)
  z35(i)=z5(i)-z3(i)
  z46(i)=z6(i)-z4(i)
  aj1(i)=x17(i)+x28(i)-x35(i)-x46(i)
  aj2(i)=y17(i)+y28(i)-y35(i)-y46(i)
  aj3(i)=z17(i)+z28(i)-z35(i)-z46(i)
  a17(i)=x17(i)+x46(i)
  a28(i)=x28(i)+x35(i)
  b17(i)=y17(i)+y46(i)
  b28(i)=y28(i)+y35(i)
  c17(i)=z17(i)+z46(i)
  c28(i)=z28(i)+z35(i)
  aj4(i)=a17(i)+a28(i)
  aj5(i)=b17(i)+b28(i)
  aj6(i)=c17(i)+c28(i)
  aj7(i)=a17(i)-a28(i)
  aj8(i)=b17(i)-b28(i)
  aj9(i)=c17(i)-c28(i)
enddo
return
end
```

grep swpf foo.s:

```
 #<swpf> Loop line 44 wasn't pipelined due to register
allocation blues.
 #<swpf>
```

Loop Splitting
(continued)

```
do i=lft,llt
  x17(i)=x7(i)-x1(i)
  x28(i)=x8(i)-x2(i)
  x35(i)=x5(i)-x3(i)
  x46(i)=x6(i)-x4(i)
  y17(i)=y7(i)-y1(i)
  y28(i)=y8(i)-y2(i)
  y35(i)=y5(i)-y3(i)
  y46(i)=y6(i)-y4(i)
  z17(i)=z7(i)-z1(i)
  z28(i)=z8(i)-z2(i)
  z35(i)=z5(i)-z3(i)
  z46(i)=z6(i)-z4(i)
enddo
do i=lft,llt
  aj1(i)=x17(i)+x28(i)-x35(i)-x46(i)
  aj2(i)=y17(i)+y28(i)-y35(i)-y46(i)
  aj3(i)=z17(i)+z28(i)-z35(i)-z46(i)
  a17(i)=x17(i)+x46(i)
  a28(i)=x28(i)+x35(i)
  b17(i)=y17(i)+y46(i)
  b28(i)=y28(i)+y35(i)
  c17(i)=z17(i)+z46(i)
  c28(i)=z28(i)+z35(i)
enddo
do i=lft,llt
  aj4(i)=a17(i)+a28(i)
  aj5(i)=b17(i)+b28(i)
  aj6(i)=c17(i)+c28(i)
  aj7(i)=a17(i)-a28(i)
  aj8(i)=b17(i)-b28(i)
  aj9(i)=c17(i)-c28(i)
enddo

return
end
```

C Loops

Pointers limit dependency analysis

Array notation shows independence

Use scalar loop indices:

```
for (i=0; i<(*pn); i++) {
        .
        .
        .
}
```

may not software pipeline, whereas

```
for (i=0; i<n; i++) {
        .
        .
        .
}
```

may.

−OPT:alias=name

Specify the pointer aliasing model to be used. If name is **any**, then the compiler will assume that any two memory references may be aliased unless it can determine otherwise (the default). If name is **typed**, the ANSI rules for object reference types (Section 3.3) are assumed − essentially, pointers of distinct base types are assumed to point to distinct, non−overlapping objects. If name is **unnamed**, pointers are also assumed never to point to named objects. **Finally, if name is *restrict*, distinct pointers are assumed to point to distinct, non−overlapping objects.** This option is unsafe, and may cause existing C programs to fail in obscure ways, so it should be used with extreme care.

Live Dangerously

–OPT:IEEE_arithmetic=n

Specify the level of conformance to IEEE 754 floating point arithmetic roundoff and overflow behavior. At level 1 (the default), do no optimizations which produce less accurate results than required by IEEE 754. At level 2, allow the use of operations which may produce less accurate inexact results (but accurate exact results) on the target hardware. Examples are the recip and rsqrt operators for a MIPS IV target. **At level 3, allow arbitrary mathematically valid transformations, even if they may produce inaccurate results for IEEE 754 specified operations, or may overflow or underflow for a valid operand range. An example is the conversion of x/y to x*recip(y) for MIPS IV targets.** See also roundoff below.

–OPT:roundoff=n

Specify the level of acceptable departure from source language floating point roundoff and overflow semantics. At level 0 (the default at optimization levels -O0 to -O2), do no optimizations which might affect the floating point behavior. At level 1, allow simple transformations which might cause limited roundoff or overflow differences (compounding such transformations could have more extensive effects). At level 2 (the default at optimization level -O3), allow more extensive transformations, such as the execution of reduction loop iterations in a different order. **At level 3, any mathematically valid transformation is enabled. Best performance in conjunction with software pipelining normally requires level 2 or above, since reassociation is required for many transformations to break recurrences in loops.** See also IEEE_arithmetic above.

Use pixie to Identify Cache Problems

Basic–block counting profiling:

Counts the number of cycles the program executes without accounting for cache misses, bank conflicts

Provides sorted list of time spent in each subroutine

Works on MP programs, too

Comparison with prof output shows where time is being spent in memory operations

No need to recompile or re–link, just run pixie (program cannot be linked –p)

```
% pixie program          (generates program.Addrs
                          and program.pixie)
% setenv LD_LIBRARY_PATH .
% program.pixie          (generates program.Counts)
% prof -pixie program
```

pixie Output

```
--------------------------------------------------------------------------------
Profile listing generated Thu Dec  1 11:18:22 1994
    with:       prof -pixie adi2
--------------------------------------------------------------------------------

Total cycles    Total Time  Instructions Cycles/inst    Clock    Target
   200761444        2.677s     253383589       0.792  75.0MHz    R8000

    32669082: Total number of Load Instructions executed.
   160627148: Total number of bytes loaded by the program.
    23709732: Total number of Store Instructions executed.
   113646670: Total number of bytes stored by the program.

        1065: Total number nops executed in branch delay slot.
    15966876: Total number conditional branches executed.
     8697925: Total number conditional branches actually taken.
         117: Total number conditional branch likely executed.
          30: Total number conditional branch likely actually taken.

           0: Total cycles waiting for current instr to finish.
   175244572: Total cycles lost to satisfy scheduling constraints.
   130814226: Total cycles lost waiting for operands be available.

--------------------------------------------------------------------------------
*  -p[rocedures] using basic-block counts.                                     *
*     Sorted in descending order by the number of cycles executed in each      *
*     procedure. Unexecuted procedures are not listed.                         *
--------------------------------------------------------------------------------

        cycles(%)  cum %     secs    instrns    calls procedure(file)

  37257216(18.56)  18.56     0.50   44040192    32768 ZSWEEP(adi2:.../adi2.f)
  37257216(18.56)  37.12     0.50   44040192    32768 YSWEEP(adi2:.../adi2.f)
  37257216(18.56)  55.67     0.50   44040192    32768 XSWEEP(adi2:.../adi2.f)
  31457280(15.67)  71.34     0.42   39845888  2097152 rand_(/usr/lib64/libftn.so:.../rand_.c)
  31457280(15.67)  87.01     0.42   33554432  2097152 irand_(/usr/lib64/libftn.so:.../rand_.c)
  23134917(11.52)  98.54     0.31   40027674      128 ADI(adi2:.../adi2.f)
   2049202( 1.02)  99.56     0.03    5982424        1 ADI.PREGION1(adi2:.../adi2.f)
    727967( 0.36)  99.92     0.01    1522211        2 ADI.PREGION0(adi2:.../adi2.f)
     69892( 0.03)  99.95     0.00     162966      346 _sinitlock(/usr/lib64/libc.so.1:.../ulocks.c)
     26132( 0.01)  99.97     0.00      43300      352 _lmalloc(/usr/lib64/libc.so.1:.../amalloc.c)
```

Cache Strategies: Maximize Locality

Instead of accessing across rows

```
do i = 1, n
   do k = 1, n
      do j = 1, n
         c(i,k) = c(i,k) + a(i,j)*b(j,k)
      enddo
   enddo
enddo
```

try to access down columns

```
do k = 1, n
   do j = 1, n
      do i = 1, n
         c(i,k) = c(i,k) + a(i,j)*b(j,k)
      enddo
   enddo
enddo
```

For C, the opposite order is appropriate

```
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      for (k=0; k<n; k++) {
         c[i][k] += a[i][j]*b[j][k];
      }
   }
}
```

Cache Thrashing and Array Padding

Conflicting arrays can cause severe thrashing in caches, especially direct−mapped.

```
      program copy
      implicit double precision (a−h, o−z)
      parameter (max=1024*1024)
      real*4 t(2), dtime
      common /a/ a(max), b(max)

      tm = dtime(t)
      do i = 1, max
         a(i) = b(i)
      enddo
      tm = dtime(t)

      write(6,1) tm
    1 format(1x,'Time = ',f5.3,' seconds')
      end
```

Padding between arrays or changing declared length avoids the mapping conflict

Because arrays are an exact multiple of cache size and are forced back−to−back in COMMON, corresponding array elements map into the same cache location.

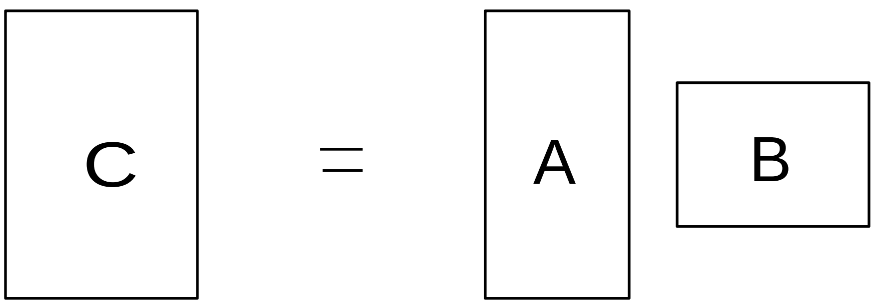
With Power Challenge's associative caches, severe thrashing does not occur *in this example*

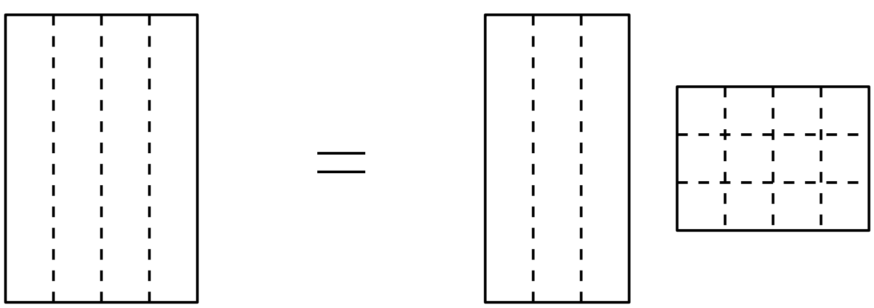
With the 2−way or 4−way set associative caches, up to 2 or 4 such conflicting references can be in cache together.

Cache Blocking

If an array doesn't fit entirely in the cache,
try to block it into pieces that do:

Example: Matrix multiply





Matrix transpose is another operation that must
be cache−blocked for good efficiency.

Memory Bandwidth

- Unrolling of loops may demonstrate the potential for data reuse.

```
      DO   10,I = 1,IMAX
        DO   10,N = 1,5
          A(N,I) = A(N,I) • B(I)
   10 CONTINUE
```

- Combining loops may uncover the potential for data reuse.

```
      DO   10,I = 1,IMAX
        A(I) = A(I) + B(I)
   10 CONTINUE
      DO   20,I = 1,IMAX
        C(I) = C(I) – B(I)
   20 CONTINUE
```

- Unrolling of loops may allow one to eliminate unnecessary or duplicate instructions resulting from prior vector optimizations.

R10K

fp exceptions work!

setenv TRAP_FPE_off

- Old compiler default was mips4
- New compiler default is mips2
- Use -mips4

R10K

has hardware
event counters

perfex -a a.out
counts all events

including
cache misses
TLB misses

Don't forget
to try
-O2
may be faster
than -O3

Summary of Uniprocessor Tuning Techniques

1. Get to top optimization level: **`−O3 −mips4`**

2. Use fast libraries: **`−lfastm −lcomplib.sgimath`**

3. Allow optimizations that affect roundoff or the last bit of precision:

   `−OPT:roundoff=3:IEEE_arithmetic=3`

3. Try getting improved SWP code by examining "love letters" in listing files and trying for lower cycle counts with:

   - *ivdep* directive/pragma
   - inlining
   - outer loop unrolling, ...

4. Make code as cache−friendly as possible:

   - Stride−1 inner loops
   - Fuse loops to get vector reuse, if necessary
   - Nest loops to access multidimensional arrays contiguously, Inner−to−outer loops traverse leftmost−to−rightmost indices (FORTRAN) or rightmost−to−leftmost indices (C)
   - Pad power−of−2 dimensions to alleviate cache−thrashing
   - Block large matrix operations for cache